LEVEL II (12)

ARPA Order Number:	3079.4
Name of Contractor:	President and Fellows of Harvard College
Effective Date of Contract:	October 1, 1977
Contract Expiration Date:	December 31, 1981
Reporting Period:	October 1, 1978-December 31, 1979
Contract Number:	N00039-78-G-0020
Principal Investigator:	Thomas E. Cheatham, Jr.
Short Title of Work:	"Refinement of an Abstract Model of MSG"
Author:	William R. Bush
Date:	April, 1980

Sponsored by
Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 3079.4
Monitored by Naval Electronic Systems Command
Under Contract #N00039-78-G-0020

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A092969	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Refinement of an Abstract Model of MSG.		5. TYPE OF REPORT & PERIOD COVERED 9 technical report. 1 Oct 78-31 Dec 79
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 William R./Bush		8. CONTRACT OR GRANT NUMBER(s) 15 N00039-78-G-0020, ARPA Order-3079
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Research in Computing Technology Harvard University Cambridge, Massachusetts 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 12 25
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Naval Electronic Systems Command Washington, D.C. 20360		12. REPORT DATE 11 Apr[redacted]80
		13. NUMBER OF PAGES twenty-two
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

unlimited

18. SUPPLEMENTARY NOTES

software development	software maintenance
program families	stepwise refinement
National Software Works	MSG

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the second phase of an experiment designed to demonstrate techniques for software development and evolution. The experiment involves the production of a family of functionally similar systems on dissimilar host computers with markedly different operating systems. The basic technique used is machine-assisted stepwise refinement from an abstract model program that embodies the desired characteristics of the family members without overconstraining the individual implementations.

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407 188

JOB

Abstract cont.

The example system developed in this experiment is MSG, the interprocess communication component of the National Software Works. Our experiment consists of producing an abstract model of MSG and realizing an instance of it, to study the techniques of abstraction and refinement and to test methods for managing the refinement process.

The first phase of the project, creation of an MSG model, is discussed in a previous technical report for this contract. "Abstract Model of MSG," by Glenn H. Holloway, William R. Bush and George H. Mealy. The second phase, refinement of the model, is described in this report. The first part of the report summarizes our work. The second part describes our approach to program development. The third part discusses our experiment with MSG.

Abstract

This report describes the second phase of an experiment designed to demonstrate techniques for software development and evolution. The experiment involves the production of a family of functionally similar systems on dissimilar host computers with markedly different operating systems. The basic technique used is machine-assisted stepwise refinement from an abstract model program that embodies the desired characteristics of the family members without overconstraining the individual implementations.

The example system developed in this experiment is MSG, the interprocess communication component of the National Software Works. Our experiment consists of producing an abstract model of MSG and realizing an instance of it, to study the techniques of abstraction and refinement and to test methods for managing the refinement process.

The first phase of the project, creation of an MSG model, is discussed in [Model]. The second phase, refinement of the model, is described in this report. The first part of the report summarizes our work. The second part describes our approach to program development. The third part discusses our experiment with MSG.

Summary

An experiment in software development has been undertaken at Harvard's Center for Research in Computing Technology. Its purpose is to test techniques intended to reduce the cost and enhance the reliability of software development and maintenance.

The experiment involves the production of a system of moderate size and complexity that has been implemented on several different types of computers by conventional means. The example system is thus actually a family of programs sharing a common specification but having a number of markedly different implementations.

Our hypothesis is that the development and maintenance of a program family can be made less costly and more dependable if the individual family members are derived from a common ancestor, which we call the abstract model of the family.

Concrete program instances are derived by a sequence of refinements from the abstract model. Each refinement encapsulates related design decisions that distinguish a class of concrete instances. A refinement may give definition to a procedure, a data structure, or a control pattern left unbound at the abstract level, or it may modify or augment such constructs. The program family thus forms a tree, with the abstract model at the root, groups of refinements as branches, and concrete instances as leaves.

We have developed, and are continually improving, tools that aid developers by maintaining the family tree in a data base and by mechanizing the task of applying refinement sequences. With these tools, a new member of an existing family is less expensive to produce than one developed separately, because the basic model and most of the refinements that yield the new version will be shared with existing instances. Moreover, modifications to broad subfamilies can be effected in unison, by applying altered refinements that reflect revised design decisions and by simply reapplying those not affected.

The specific system we are experimenting with, called MSG, is an ARPANET interprocess communications handler used by the National Software Works. It runs under TENEX, TOPS-20, OS/MVT, Multics, and UNIX. MSG is thus a good paradigm for our experiment, since it is a moderately complex family of programs that must

function similarly on highly disparate hosts.

Our goal was to test our abstraction and refinement methodology by producing, from our abstract model, TENEX and UNIX instances and comparing them to the existing conventionally produced implementations. We investigated refining to those systems but did not produce instances for them. For various reasons a related project was not completed, which we needed to translate our refined instances, written in interpreted EL1, into a lower-level compiled language allowing stand-alone execution. Nonetheless, we have produced a refined version that runs under the ECL interpreter extended by a package that supports multiple processes. The interpreted version has been used to debug the model, running it with debugging tools in the benign environment of the interpreter.

We found that, given a carefully designed abstract model, it was relatively straightforward to produce a refined instance. Having anticipated refining while working on the model, we were able to develop refinements without difficulty, and no significant changes in the model were needed after refinement development started.

Theory

Methodology

There is much concern over the cost of software design, implementation, and maintenance. This has stimulated research in areas from language design to program verification. The goal of our work at Harvard is a programming methodology, supported by appropriate tools, that makes software development and maintenance more efficient by promoting clear, high-level design, and by having different implementations of the same system share a single high-level description. The basic technique is initially to write the high-level program, focussing on the fundamental aspects of the problem to be solved. Later, details are added. These details are added in stages, so that later details are more efficiency-related and computer-type-specific. Thus the details necessary to produce an efficient, running program do not intrude on the basic specification of the algorithm. For example, at some point the high-level program may sequentially examine the elements of a set, in which case the iterator

```
ForEach E In S Do ... End;
```

may be used as an intuitively clear construct. Later, implementation details of ForEach are supplied in terms of conventional primitives, for example, a FOR loop. The point is to layer the programming process, keeping the fundamental program clear and general, and adding groups of related implementation details as needed. We call the high-level program the *abstract model*: 'abstract' because it is high-level, and 'model' because it is a structure that forms the basis for actual running programs. Note that 'abstract' does not imply a formal mathematical object. We call the details added to the model *refinements*.

The clarity of the model and the layering of refinements aid maintenance. Programs continue to be used many years after they are written, and, as circumstances change, modifications are required. Such modifications are often not made by the implementers. Thus changes tend to be ad hoc and incorrect and do violence to the basic structure of the programs, making them less efficient and harder to comprehend. In contrast, the hierarchical form of the abstract model and refinements, separating basic structure from implementation details, makes the structure clear and more easily understood. Furthermore, grouping related refinements together encapsulates and localizes implementation decisions, making it easy to review and change them. Basically, the program itself reflects the hierarchical nature of its design and implementation.

Modelling and refinement also are efficient means of producing and maintaining program families. A *family* of programs shares a common specification but has a number of different implementations, or instances. Examples of program families are compilers of the same programming language for different computers, operating systems providing similar environments on different physical configurations, and communications processors observing a common protocol while running on different types of host machines. The abstract model embodies the family specification, and different groups of instance-specific refinements applied to it produce different instances. Some instances may share refinements. Thus the family takes the form of a tree, with the abstract model as the root, groups of refinements as branches, classes of instances sharing refinements as interior nodes, and individual instances as leaves. Once the abstract model is written, a new instance may be produced cheaply by writing the appropriate refinements. Further economies are realized through shared refinements. Maintenance is greatly simplified since changes in the specification -- the abstract model -- are reflected at once in all instances. A single algorithmic specification is an effective means of insuring common behavior among all family members, compared to a functional specification implemented independently on different hosts.

Technique

Our models and refinements are written in EL1 [ECL], which facilitates abstraction and refinement in two major ways.

First, EL1 is an extensible language, with user-definable modes and syntax. User-defined modes are built out of basic ones and may include user-provided functions invoked upon generation, conversion, assignment, selection, printing, and determination of dimensions. Syntax can be extended by giving any symbol one of several parsing properties: prefix, infix, matchfix (two symbols paired, for example, { and }), or synfix (one symbol given the properties of another, for example, ForEach given the properties of FOR). With these facilities modes and syntax are used in the abstract model, and their underlying functionality is defined and augmented in refinements.

Second, the ECL system, which implements the EL1 language, is an interpretive one, with programs represented internally as list structure. This makes easy the automatic analysis and modification of programs by other programs. For refinement we use a pattern-driven transformation tool that replaces one syntactic pattern, possibly including match variables and qualified by a predicate, with another. This tool substantially increases our ability to modify and augment the abstract model.

The refinement process is managed by the Program Development System [Refinement] [PDS]. The PDS provides the environment for defining and modifying an abstract model and its refinements, and for producing refined instances.

The model is defined in terms of *modules*, collections of related entities. An *entity* is a procedure, mode, or data object. Entities may be clustered in groups within a module. Modules are insulated from one another; contact between them is defined by export and import lists.

Refinements are entities as well. They may define or augment a procedure, mode, or data object, or specify a transformation pattern and its replacement (a rewrite). They may be applied singly or in groups to an entity, to a group of entities, to all the entities in a module, or to a number of modules.

The PDS produces a program instance by taking each module of the abstract model and transforming the entities in it using the proper refinements, then merging all the modules. The PDS recognizes when an entity is modified in the course of development, so when a new version of an instance is required only those entities dependent on modified ones are retransformed.

Managing refinement is the central function of the PDS, but it also supports the programming process in other ways. It uses an editor that has knowledge of entities and EL1 structure. It supports various analysis tools (for example, one that locates undefined identifiers). It aids interpretive debugging and compilation. In these ways it extends the ECL system, supplying structure not found in the language.

Our technique, then, uses not just data abstraction, extended syntax, or pattern-driven transformation, but all three, in a sophisticated programming system possessing substantial knowledge of the programming language and the means of abstraction and refinement. A program, in the form of an abstract model and refinements, embodies the hierarchical process of design and implementation. Our method is not a discipline that keeps people from writing bad programs, but an environment in which it is easier to write clear and maintainable ones.

Practice

Implementing A Program Family

In order to develop and test our methodology and tools we have used them to produce a family of programs. The system we have chosen is a network interprocess communications handler that runs on various types of computers. It is apt for our purposes since it must function similarly on different hosts according to a common specification.

The system is MSG, which runs under various operating systems on various computers on the ARPANET. It starts and manages the local processes using it, and routes messages between them and other processes on other hosts. Interhost-interprocess messages are multiplexed, using a common protocol, over network connections between MSG instances. The protocol includes items that facilitate flow control, so that the burden of buffering messages can be distributed between sender and receiver when traffic is heavy. The protocol also provides for interprocess signalling of exceptional events and for establishing direct network connections between processes. MSG is thus complex enough to provide a serious test of our tools. We wrote an abstract model of MSG (described fully in [Model]), produced an instance that runs under an extension of the ECL interpreter [CI], and investigated refining to TENEX and UNIX.

We implemented MSG in three phases. First, we defined the overall structure of the abstract model, dividing it into major functional units, and designed a few basic abstractions used throughout. Our goals were host-independence, clarity, and refineability. Second, we coded a procedural embodiment of the MSG Design Specification [MSG], using the results of the first phase as foundation. Our main concerns were understanding and correctly transcribing the details of the MSG protocol. While this involved much work (a substantial body of code was generated), it was a relatively straightforward and low-level enterprise. Third, we produced the refinements necessary to realize a working MSG. At this point host-dependent issues were paramount. This stage was also relatively easy, since we had considered refinement from the first. The remainder of this section summarizes the results of the first phase, describing in general terms the model and its primary abstractions.

The basic function of MSG is to respond to local processes on the one hand and the network on the other, routing data between them. The form of our abstract model follows this functional division, having three major parts, each a module – one handles local processes, one handles the network, and one routes data, via queues, between the first two. Other modules provide global definitions, initialization, and error recovery.

All parts of MSG share the data base of queues that the routines in the queue module maintain. This shared data base gives us great flexibility, for it means that intra-MSG process boundaries need not be specified in the model. This allows us to tailor process structure to particular hosts and makes possible the diverse structures found in figures 1, 2, and 3. In fact, without major effort we can test different process structures on the same host to find an optimal one.

Clarity in the model is promoted by abstract queue managers and iterators, which manipulate the data base. These constructs are easily understood at the abstract level without reference to their implementation, and keep from the model the details of queue threads and their use. The abstractions are also few in number and uniform throughout the model. Examples of them and their refinements are found in the next section.

The most difficult construct to define was a host-independent mechanism for interprocess communication and network input/output. It had to be simple and clear, yet refine to the varied primitives of our target hosts. Refineability was a crucial test; some designs failed because they were too general to be refined at all, and others because they were too specific to be refined on some hosts. Our final construct, called a channel, is a union of the central features of the actual primitives we must use. It is a named data path, possibly bi-directional, that, when used, may block or interrupt the processes using it. It may be opened and closed, and data may be sent and received over it. It refines into concrete CI channel primitives, TENEX interprocess core mapping and network access functions, and UNIX pipes and ports.

Examples Of Abstraction And Refinement

The following representative abstract constructs and their refinements illustrate the technique used in the implementation of MSG. To keep the examples brief, some details in the actual refinements are omitted. The refinements are common to all instances (unlike others that are lower-level; for example, those for channels) and are collected in one module.

Consider the abstract primitives

```
Enqueue Message AtFrontOf InputMessageQueue;
```

and

```
ForEach Message FromFrontOf OutputMessageQueue Do ... End;
```

They are intuitively clear – one inserts a queue element at the front of a queue, and the other iterates over a queue, sequentially producing each element in it – and are used in this form in the abstract model. They are not, however, part of the EL1 base language. They are transformed by refinement to executable code.

A queue can be implemented in many ways. None need concern us at the abstract level. During refinement, however, we select the best implementation for MSG, taking advantage of the way MSG queues behave. Since any queue element can appear at most once in each of a number of known queues, we add, by refinement, in each queue element a queue thread for each known queue, and make the queue primitives use the proper thread.

More specifically, a **MessageBlock** is a data structure used in the abstract model (its fields include the text of an interprocess message and the names of the source and destination processes). A **MessageHandle**, of which **Message** is an instance, is a pointer to a **MessageBlock**. A **MessageHandle** may appear in one or both of two **MessageQueues**, an **InputMessageQueue** and an **OutputMessageQueue**. With the following refinements we add queue thread fields for the two queues to **MessageBlock**, and create the queue mode.

```
MessageBlock Has
    AddedFields(InputMessageQueue:MessageHandle,
                OutputMessageQueue:MessageHandle);
MessageQueue Is STRUCT(Front:MessageHandle, Count:INT);
```

Consider the Enqueue primitive. It must splice a queue element into a queue by setting the front of the queue to the new element and the queue's thread in the new element to the element that was at the front. The following rewrite defines how the abstract construct is to be transformed. (Rewrites have the form **search-pattern <->** **replacement-pattern**, with pattern variables in *italics* and pattern predicates in *italics*.)

```
Enqueue element AtFrontOf queue
    <->
BEGIN
   queue.Count <- queue.Count + 1;
   element.queue <- queue.Front;
   queue.Front <- element;
END;
```

When applied to the abstract instance above the rewrite produces the following code:

```
InputMessageQueue.Count <- InputMessageQueue.Count + 1;
Message.InputMessageQueue <- InputMessageQueue.Front;
InputMessageQueue.Front <- Message;
```

The ForEach primitive must scan the length of a queue, starting at its front, obtaining each successive element from the thread field of its predecessor. The following rewrite generates the appropriate code.

```
ForEach element FromFrontOf queue Do body End
    <->
BEGIN
   DECL element:MessageHandle BYVAL queue.Front;
   TO queue.Count
      REPEAT
         body;
         element <- VAL(element).queue;
      END;
END;
```

The actual refinement of ForEach is considerably more complicated – it involves retarded pointers and storage management for insertion and deletion within the loop.

Even though we are concerned with clarity, and not efficiency, at the abstract level, we can still produce efficient, optimized code by refinement. For example, we implement sets in terms of both sequences and lists, and choose an implementation for a particular set depending on how the set is used. Similarly the abstract set iterator

```
ForEach E In S Do ... End;
```

has refinements defined for both set representations. The refinement applied in a given instance is based on the mode of the set being refined. The sequence refinement appears as

```
ForEach element In set Do body End
    Where set HasMode SEQ(mode)
        <->
FOR ForEachIndex TO LENGTH(set)
    REPEAT
        DECL element:mode LIKE set[ForEachIndex];
        body;
    END;
```

and the list refinement as

```
ForEach element In set Do body End
    Where set HasMode PTR(mode)
        <->
BEGIN
    DECL ForEachPointer:PTR(mode) BYVAL set;
    REPEAT
        ForEachPointer = NIL => NOTHING;
        DECL element:mode LIKE VAL(ForEachPointer);
        body;
        ForEachPointer <- element.set;
    END;
END;
```

The rewrites are applied only to patterns fulfilling the requirements of the *Where* predicates.

Producing An Instance

The process of transforming the MSG abstract model via refinements into a running program is managed for us by the PDS and consists of four steps, illustrated in figure 4.

First, each individual abstract module (for example, the one containing the queuing routines) is transformed with the host-independent refinements for that module (which are themselves gathered in a module). The result is a lower-level, host-independent version of the abstract module, shared by all MSG instances. For example, queue threads are added to data structures at this point.

Second, in a similar manner each host-independent module is transformed by module-specific, host-dependent refinements, producing a host-dependent module. For example, channel refinement is done at this point.

Third, each host-dependent module is made into runnable code by importing from other specified host-dependent modules globally available entities needed to complete the module's realization. For example, external references are resolved at this point.

Fourth, all runnable modules are combined into one executable program.

Not all modules go through both steps one and two. Some have no host-independent version and others no host-dependent one. For example, the initialization module is taken from abstract code directly by step two to its host-dependent form and then by step three to runnable code, while the queue module is taken from its host-independent form after step one directly to runnable code by step three. On the other hand, both the local process and network modules go through all four steps.

Steps one and two are actually the parts of a larger single process, the application of module-specific refinements. The PDS simply applies a series of refinement modules in order to an abstract module. The nature and number of module-specific refinement steps depends on the program family, each step encapsulating related design decisions. In general, higher-level refinements are grouped together and applied before lower-level ones, promoting clarity (keeping levels of abstraction distinct) and economy (refinements shared among instances exist in one module only). The more family members there are, the more steps there tend to be, due to refinement sharing among members. For example, if we produced a TENEX instance, we would divide step two into two steps, first applying host-dependent refinements common to CI and TENEX, and then applying those particular to each system.

The refinements applied in steps one and two remain for global application in step three. Thus the low-level definitions and rewrites, both host-independent and host-dependent, of a given module are available to other modules in the third step. In this step, for example, global references to the queue thread fields of a **Message Block** are properly resolved, and some common channel rewrites are applied to several modules.

The PDS automatically performs all four refinement steps for us. We simply provide it with modules and refinement instructions (what refinements should be applied when to which entities), and it generates an MSG instance. It guarantees that names local to a module are hidden from the global environment (step three), that data objects are initialized properly (step four), and that the load order of objects in the final program is correct (step four).

Conclusions

After we had given considerable thought to the form of our abstract model and a few constructs widely used throughout it, we found it straightforward to code the model and refine it. This ease was the reward for our hierarchical approach to design, supported by our tools. We could give adequate attention to the high-level issues of program structure, meaningful modes and operators, and host-independence without getting mired down in details. Furthermore, these issues found expression, not in design documents used for reference, but in actual primitives usable in coding. Since we considered initially the broad, difficult problems of design and had a proper medium for expressing our solutions, our implementation effort was relatively easy and contained few surprises. Nor did abstraction exact a great run-time price, because our refinement tools and techniques allow us to produce efficient code.

Implementation was made easier, and understanding our model more difficult, by our decision always to produce abstractions that were obviously refineable. We could have produced a model in a clearer form, for example, in the form of a finite state machine, but refinement would have been problematic. Still, we found it easier to read unfamiliar model code than code produced conventionally.

Our technique provides the greatest benefit when programming is undertaken in a hierarchical manner. It encourages comprehension before implementation. Nonetheless, implementation of any large new system requires experimentation and backtracking. Our tools facilitate such iterative development by making the system easy to dismantle and reconstruct.

Most of our effort was spent designing and coding the model; substantially less was spent on refinement. Having found realization of an instance relatively cheap, given an abstract model, we recommend the abstraction-refinement paradigm as an economical one for development of a program family. In terms of volume, most of the MSG code is in the abstract model, two and a half times as much being there as in the refinements. The general and host-specific refinements are of equal volume. By type, the refinement entities are composed of roughly equal numbers of rewrites and of definitions of low-level procedures, modes, and data objects. There are about a third as many augments to existing definitions as there are new ones. Most of the rewrites are of general applicability; only about a tenth of them are substantial host-specific modifications to a single procedure.

We also recommend the abstraction-refinement paradigm for program families because the abstract model serves as a complete, universal, operative definition of family behavior. For example, the MSG Design Specification does not fully define error handling, so that some conditions may or may not be errors, depending on the interpretation of the implementers. Thus a robust implementation must anticipate the various possible responses to a possible error. Having all implementations descend from a single procedural definition eliminates this problem.

Programming is at once easy (the primitives one deals with are simple and relatively few) and very difficult (building a large, correct, complex structure from those primitives requires substantial skill). Many efforts have been made to make the programming process less error-prone and more efficient. Some techniques reason about the program (verification), some develop new primitives (language design), some provide mechanisms for the programmer to define new primitives (extensibility, abstraction), and some build tools that support program development and management. Our work centers around the last two approaches, allowing and managing a hierarchy of program structure. We keep complexity and detail from program development until they are necessary.

Notes to Figures 1, 2, and 3

The ellipses are operating system processes; the arrows are channels; the dotted line delineates the boundary of the shared data base of queues.

The CI implementation is relatively unconstrained by its operating system. Thus each logically separate function is assigned its own process.

On TENEX processes are expensive, so they are minimized (under the constraint that network input is blocking – hence the several network input servers). All user processes of a given class are handled by a single server, and all network output is performed by one process.

The UNIX implementation is radically different, due to the lack of a facility for sharing core. All functions that must access the data base of queues are in one fork, with I/O functions that may block supported by ancillary forks. Scheduling of tasks within the main MSG fork is done by MSG itself.

Note to Figure 4

The ellipses are modules; the arrows are refinement steps.

Note on CI

We desired to test MSG in an environment supported by the ECL interpreter, with our debugging tools available. To accomplish this we needed the interpreter extended to provide some of the functionality of an operating system. Building on work done for a doctoral dissertation, we have implemented an extension that supports multiple processes and scheduling, timing, interrupts, completion events, and an MSG-like channel facility that subsumes I/O and interprocess communication.

Figure 1 – CI MSG

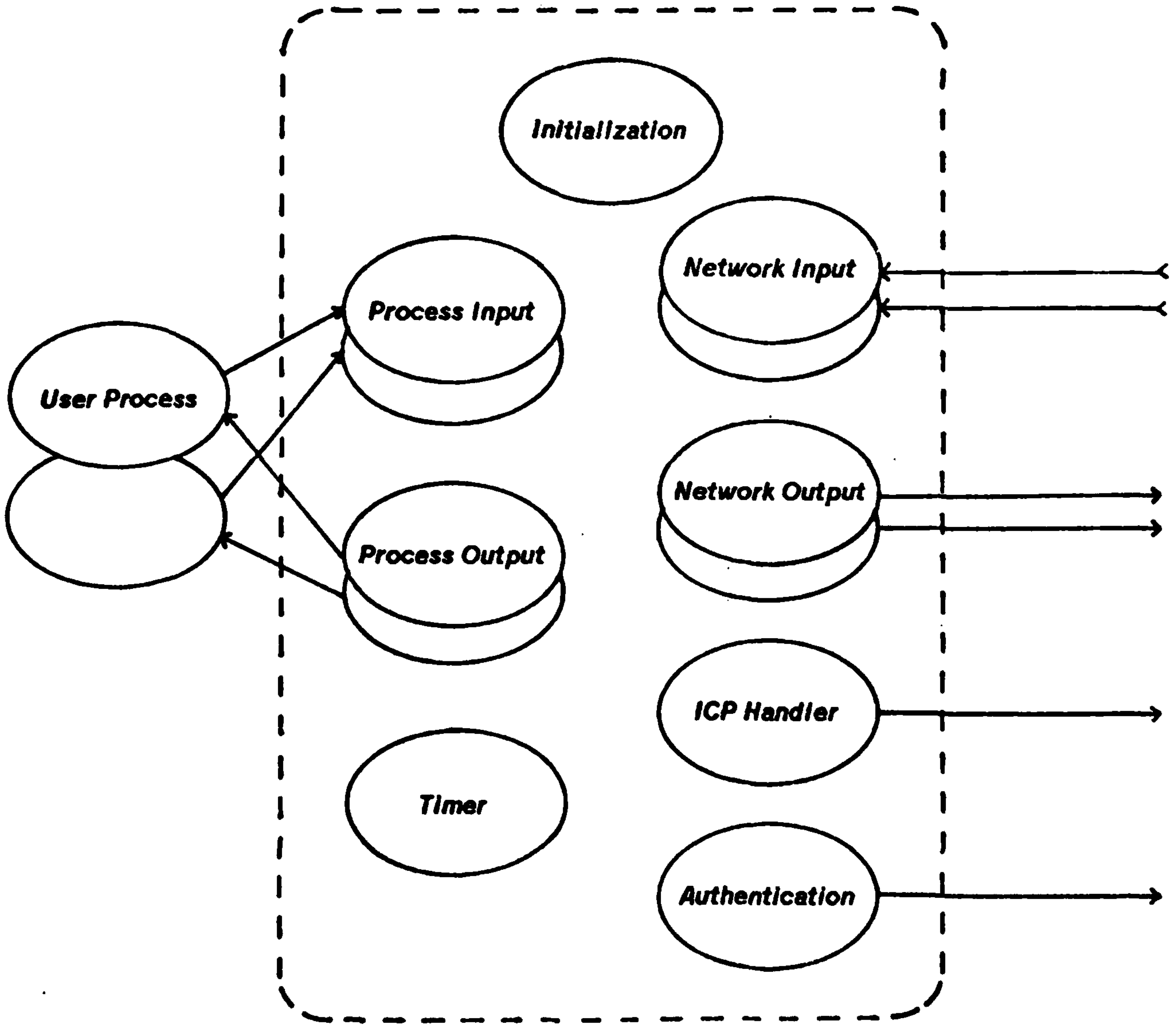


Figure 2 – TENEX MSG

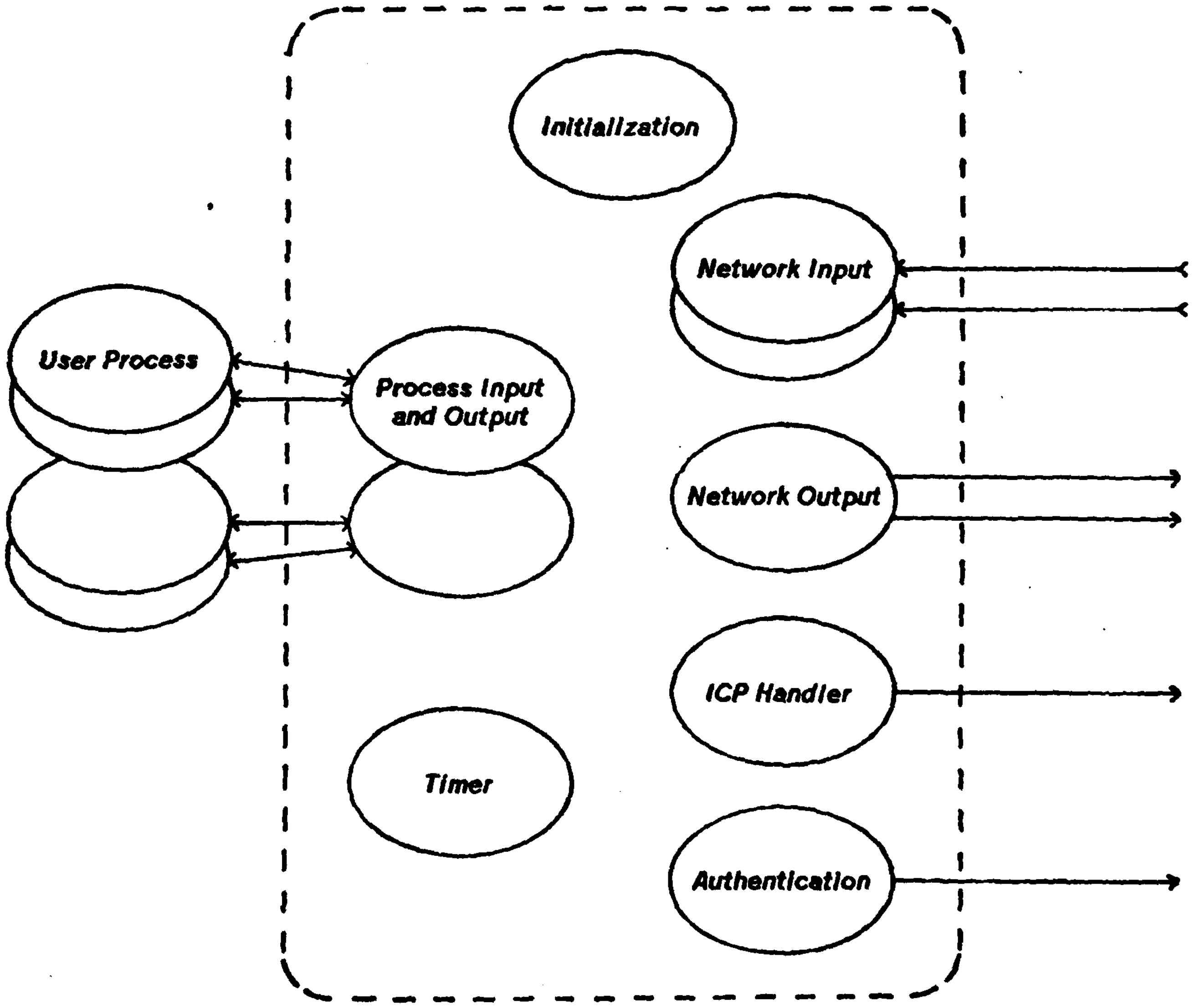


Figure 3 – UNIX MSG

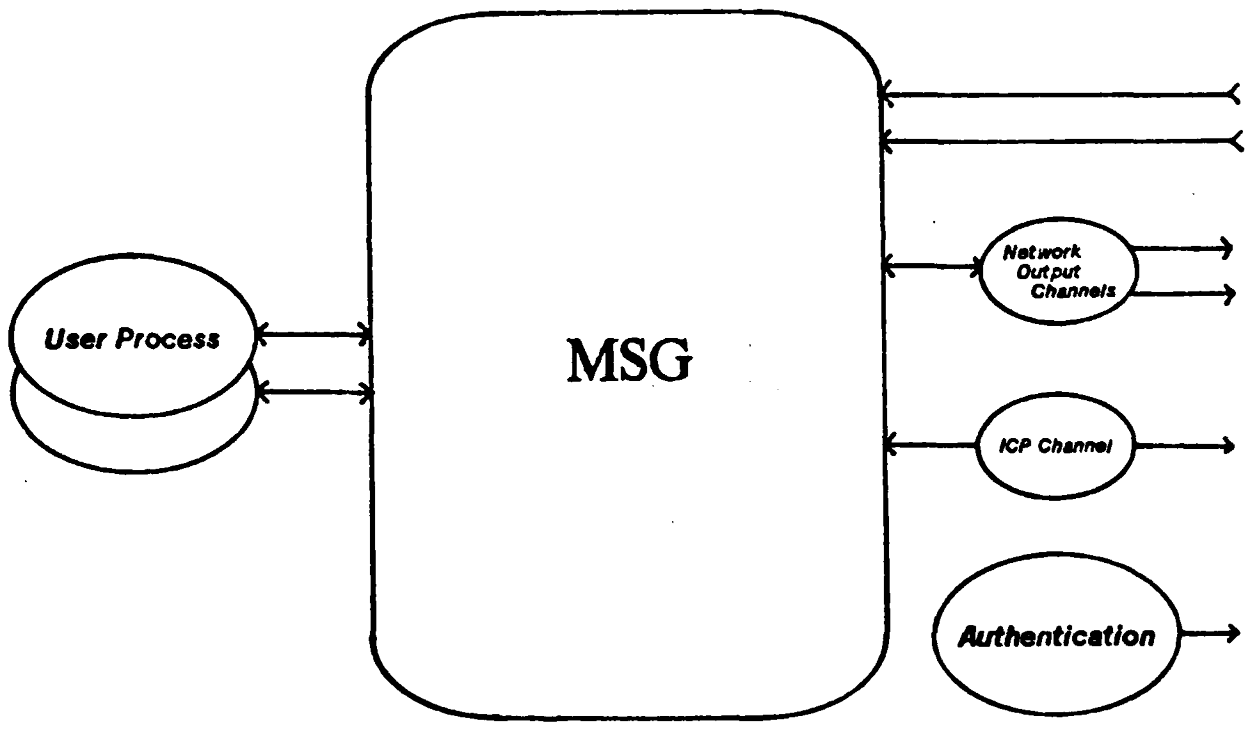
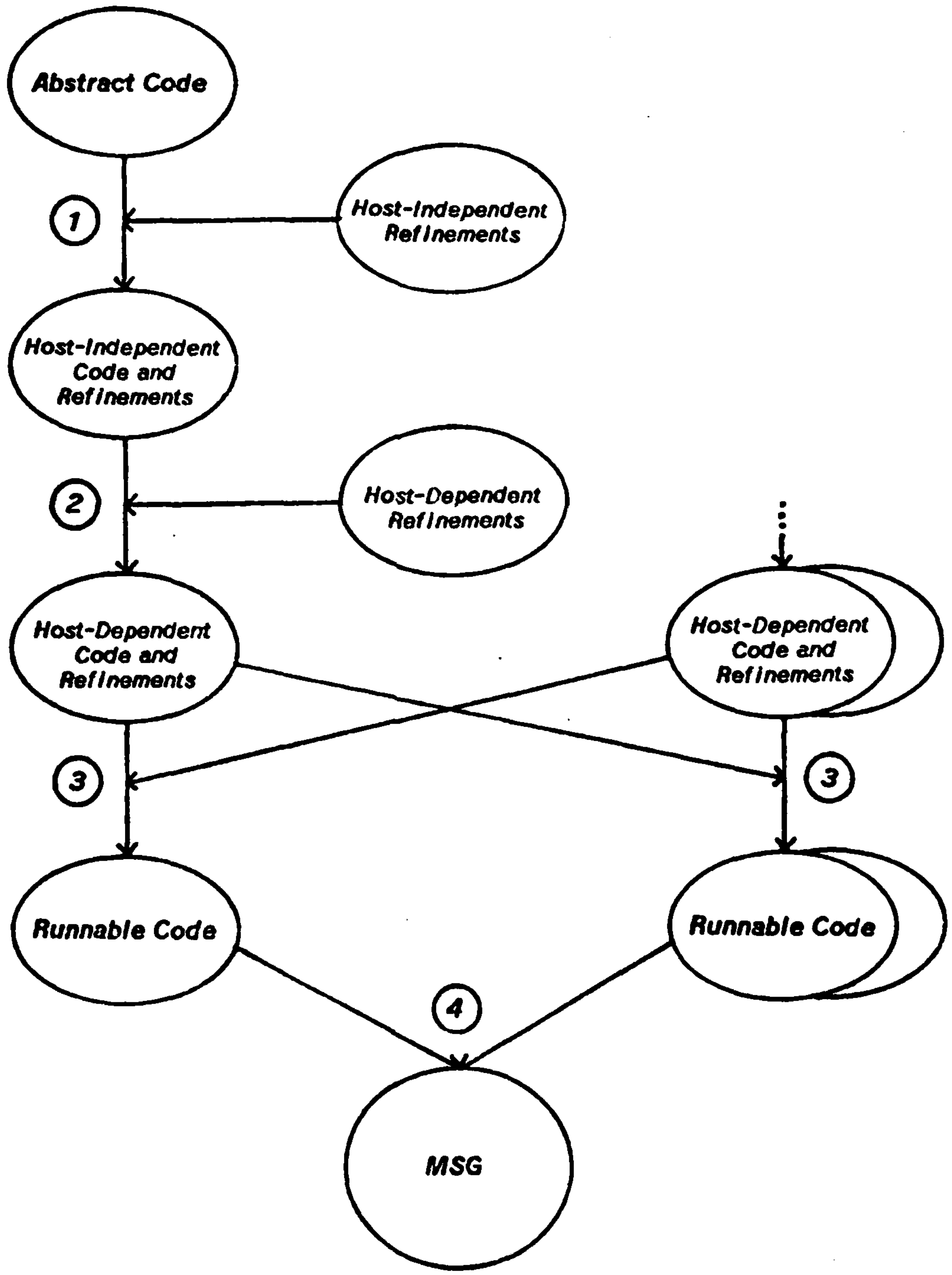


Figure 4 – Transforming MSG


Abstract Code
Host-Independent Refinements
1
Host-Independent Code and Refinements
Host-Dependent Refinements
2
Host-Dependent Code and Refinements
Host-Dependent Code and Refinements
3
3
Runnable Code
Runnable Code
4
MSG

References

[CI] *An ECL Control Interpreter.* Memorandum, Center for Research in Computing Technology, Harvard University, Cambridge MA, April 1980.

[ECL] *ECL Programmer's Manual.* Technical report TR-23-74, Center for Research in Computing Technology, Harvard University, Cambridge MA, December 1974.

[Model] *Abstract Model of MSG.* Technical report TR-23-74, Center for Research in Computing Technology, Harvard University, Cambridge MA, October 1978.

[MSG] 'MSG Design Specification', in *Third Semi-Annual Technical Report for the National Software Works.* Massachusetts Computer Associates, Wakefield MA, February 1977.

[PDS] *User's Manual for the Harvard Program Development System.* Memorandum, Center for Research in Computing Technology, Harvard University, Cambridge MA, January 1980.

[Refinement] *A System for Program Refinement.* Technical report TR-05-79, Center for Research in Computing Technology, Harvard University, Cambridge MA, August 1979.